

A Polymorphic Groundness Analysis of Logic Programs

Lunjin Lu

Department of Computer Science
 University of Waikato
 Hamilton, New Zealand
 EMail: lunjin@cs.waikato.ac.nz

Abstract. A polymorphic analysis is an analysis whose input and output contain parameters which serve as placeholders for information that is unknown before analysis but provided after analysis. In this paper, we present a polymorphic groundness analysis that infers parameterised groundness descriptions of the variables of interest at a program point. The polymorphic groundness analysis is designed by replacing two primitive operators used in a monomorphic groundness analysis and is shown to be as precise as the monomorphic groundness analysis for any possible values for mode parameters. Experimental results of a prototype implementation of the polymorphic groundness analysis are given.

1 Introduction

Groundness analysis is one of the most important dataflow analyses for logic programs. It provides answers to questions such as whether, at a program point, a variable is definitely bound to a term that contains no variables. This is useful not only to an optimising compiler that attempts to speed up unification but also to other program manipulation tools that apply dataflow analyses. Groundness analysis has also been used to improve precisions of other dataflow analyses such as sharing analysis [38, 19, 20, 6, 10, 34, 41, 24] and type analysis [28]. There have been many methods proposed for groundness analysis [32, 15, 6, 41, 11, 30, 11, 1, 2, 5, 25, 17, 7, 9, 39].

This paper present a new groundness analysis, called polymorphic groundness analysis, whose input and output are parameterised by a number of mode parameters. These mode parameters represent groundness information that is not available before analysis but can be provided after analysis. When the groundness information is provided, the result of the polymorphic groundness analysis can then be "instantiated". Consider the following program

```

p(X,Y) :- q(X,Y), X<Y, ...
q(U,U).

```

and the goal $p(X, Y)$ with mode parameters α and β being respectively the groundness of X and Y prior to the execution of the goal. The polymorphic groundness analysis infers that at the program point immediately before the built-in call $X < Y$, the groundness of both X and Y are the greatest lower bound of α and β implying that X and Y are in the intersection of the sets of terms described by α and β . The result can be instantiated when the values of α and β become available. It can also be used to infer sufficient conditions for safely removing run-time checks on the groundness of the operands of the built-in predicate " $<$ ".

The polymorphic groundness analysis is performed by abstract interpretation [12, 13, 14]. Abstract interpretation is a methodology for static program analysis whereby a program analysis is viewed as the execution of the program over a non-standard data domain. A typical analysis by abstract interpretation is monomorphic whereby input information about the program is not parameterised and the program has to be analysed separately for different input information. The polymorphic groundness analysis is one of a particular class of program analyses whereby input and output of a program analysis contain parameters which express information that is unknown before analysis but may be provided after analysis. Polymorphic program analyses have advantages over monomorphic program analyses since more general result can be obtained from a polymorphic analysis. Firstly, the result of a polymorphic analysis is reusable. A sub-program or a library program that may be used in different places need not to be analysed separately for its different uses. Secondly, polymorphic program analyses are amenable to program modifications since changes to the program does not necessitate re-analyses of the sub-program so long as the sub-program itself is not changed.

The polymorphic groundness analysis is a polymorphic abstract interpretation that formalises a polymorphic analysis as a generalisation of a possibly infinitely many monomorphic analyses [29]. It is obtained from a monomorphic groundness analysis by replacing monomorphic description domains with polymorphic description domains and two primitive operators for the monomorphic groundness analysis with their polymorphic counterparts. It is proven that

for any possible assignment of values for parameters, the instantiated results of the polymorphic groundness analysis is as precise as that of the monomorphic groundness analysis corresponding to the assignment.

An abstract interpretation framework is a generic abstract semantics that has as a parameter a domain, called an abstract domain, and a fixed number of operators, called abstract operators, associated with the abstract domain. A particular analysis corresponds to a particular abstract domain and its associated abstract operators. Usually, specialising a framework for a particular analysis involves devising an abstract domain for descriptions of sets of substitutions, called abstract substitutions, and corresponding abstract operators on abstract substitutions. One important abstract operator commonly required is abstract unification [6, 21] which mainly abstracts the normal unification. Another important abstract operator commonly required is an operator that computes (an approximation of) the least upper bound of abstract substitutions. The polymorphic groundness analysis will be presented as an abstract domain for polymorphic groundness descriptions of sets of substitutions together with its associated abstract unification operator and least upper bound operator as required by the framework in [27]. The adaptation to the frameworks in [6, 20, 31, 36] needs only minor technical adjustments since the functionalities required of the above two abstract operators by these frameworks and that in [27] are almost the same. The adaptation to the frameworks in [4, 22, 33] needs more technical work but should not be difficult because most functionalities of the abstract operators in these frameworks are covered by the functionalities of the abstract unification operator and the least upper bound operator in [6, 20, 27, 36].

The remainder of the paper is organised as follows. Section 2 introduces basic notations, abstract interpretation and an abstract interpretation framework of logic programs based on which we present our polymorphic groundness analysis. Sections 3 recalls the notion of polymorphic abstract interpretations. Section 4 reformulates the monomorphic groundness analysis and section 5 presents the polymorphic groundness analysis. Section 6 contains results of the polymorphic groundness analysis on some example programs and provides performance results of a prototype implementation of the poly-

morphic groundness analysis. Section 7 concludes the paper with a comparison with related work on groundness analysis of logic programs.

2 Preliminaries

This section recalls the concept of abstract interpretation and an abstract interpretation framework based on which we will present our polymorphic groundness analysis. The reader is assumed to be familiar with terminology of logic programming [26].

2.1 Notation

Let Σ be a set of *function symbols*, Π be a set of *predicate symbols*, \mathcal{V} be a denumerable set of variables. f/n denotes an arbitrary function symbol, and capital letters denote variables. **Term** denotes the set of *terms* that can be constructed from Σ and \mathcal{V} . t, t_i and $f(t_1, \dots, t_n)$ denote arbitrary terms. **Atom** denotes the set of *atoms* constructible from Π, Σ and \mathcal{V} . a_1 and a_2 denote arbitrary atoms. θ and θ_i denote substitutions. Let θ be a *substitution* and $\mathcal{V} \subseteq \mathcal{V}$. $\text{dom}(\theta)$ denotes the domain of θ . $\theta|\mathcal{V}$ denotes the restriction of θ to \mathcal{V} . As a convention, the function composition operator \circ binds stronger than \setminus . For instance, $\theta_1 \circ \theta_2|\mathcal{V}$ is equal to $(\theta_1 \circ \theta_2)|\mathcal{V}$. An *expression* O is a term, an atom, a literal, a clause, a goal etc. $\text{vars}(O)$ denotes the set of variables in O . The range $\text{range}(\theta)$ of a substitution θ is $\text{range}(\theta) \stackrel{\text{def}}{=} \bigcup_{X \in \text{dom}(\theta)} \text{vars}(\theta(X))$.

An *equation* is a formula of the form $l = r$ where either $l, r \in \text{Term}$ or $l, r \in \text{Atom}$. The set of all equations is denoted as **Eqn**. Let $E \in \wp(\text{Eqn})$. E is in *solved form* if, for each equation $l = r$ in E , l is a variable that does not occur on the right side of any equation in E . For a set of equations $E \in \wp(\text{Eqn})$, $\text{mgu} : \wp(\text{Eqn}) \mapsto \text{Sub} \cup \{\text{fail}\}$ returns either a most general unifier for E if E is unifiable or *fail* otherwise, where **Sub** is the set of substitutions. $\text{mgu}(\{l = r\})$ is sometimes written as $\text{mgu}(l, r)$. Let $\theta \circ \text{fail} \stackrel{\text{def}}{=} \text{fail}$ and $\text{fail} \circ \theta \stackrel{\text{def}}{=} \text{fail}$ for any $\theta \in \text{Sub} \cup \{\text{fail}\}$. There is a natural bijection between substitutions and the sets of equations in solved form. $\text{eq}(\theta)$ denotes the set of equations in solved form corresponding to a substitution θ . $\text{eq}(\text{fail}) \stackrel{\text{def}}{=} \text{fail}$.

We will use a renaming substitution Ψ which renames a variable into a variable that has not been encountered before. Let \mathcal{V}_P be the set of program variables of interest. \mathcal{V}_P is usually the set of variables in the program.

2.2 Abstract Interpretation

Suppose that we have the *append* program in figure 1.

```
append([], L, L) %C1
append([H|L1], L2, [H|L3]) ← append(L1, L2, L3) %C2
```

Fig. 1. Logic program *append*

The purpose of groundness analysis is to find answers to such questions as in the following.

If L_1 and L_2 are ground before $append(L_1, L_2, L_3)$ is executed, will L_3 be ground after $append(L_1, L_2, L_3)$ is successfully executed?

Abstract interpretation performs an analysis by mimicking the normal execution of a program.

Normal Execution To provide an intuitive insight into abstract interpretation, let us consider how an execution of goal $g_0 : append(L_1, L_2, L_3)$ transforms one program state $\theta_0 : \{L_1 \mapsto [s(0)], L_2 \mapsto [0]\}$ which satisfies the condition in the above question into another program state $\theta_3 : \{L_1 \mapsto [s(0)], L_2 \mapsto [0], L_3 \mapsto [s(0), 0]\}$. We deviate slightly from Prolog-like logic programming systems. Firstly, substitutions (program states) have been made explicit because the purpose of a program analysis is to infer properties about substitutions. Secondly, when a clause is selected to satisfy a goal with an input substitution, the goal and the input substitution instead of the selected clause are renamed. This is because we want to keep track of values of variables occurring in the program rather than those of their renaming instances. Let $\Psi(Z) = Z'$ for any $Z \in \mathcal{V}$.

The first step performs a procedure entry. g_0 and θ_0 are renamed by into $\Psi(g_0) : append(L'_1, L'_2, L'_3)$ and $\Psi(\theta_0) : \{L'_1 \mapsto [s(0)], L'_2 \mapsto [0]\}$. Then C2 is selected and its head $append([H|L_1], L_2, [H|L_3])$ is unified with $\Psi(g_0)$ resulting in $E_1 : \{L'_1 \mapsto [H|L_1], L'_2 \mapsto L_2, L'_3 \mapsto [H|L_3]\}$. Then $\Psi(\theta_0)$ and E_1 are used to compute $\theta_1 : \{H \mapsto s(0), L_1 \mapsto [], L_2 \mapsto [0]\}$ the input substitution of the body $g_1 : append(L_1, L_2, L_3)$ of C2. In this way, the initial goal g_0 with its input substitution θ_0 has been reduced into the goal g_1 and its input substitution θ_1 .

The execution of g_1 , details of which have been omitted, transforms θ_1 into $\theta_2 : \{H \mapsto s(0), L_1 \mapsto [], L_2 \mapsto [0], L_3 \mapsto [0]\}$ the output substitution of g_1 .

The last step performs a procedure exit. The head of C2 and θ_2 are renamed by Ψ , and then the renamed head $append([H'|L'_1], L'_2, [H'|L'_3])$ is unified with g_0 resulting in $E_2 : \{L_1 \mapsto [H'|L'_1], L_2 \mapsto L'_2, L_3 \mapsto [H'|L'_3]\}$. Then $\Psi(\theta_2) : \{H' \mapsto s(0), L'_1 \mapsto [], L'_2 \mapsto [0], L'_3 \mapsto [0]\}$ and E_2 are used to update the input substitution θ_0 of g_0 and this results in the output substitution $\theta_3 : \{L_1 \mapsto [s(0)], L_2 \mapsto [0], L_3 \mapsto [s(0), 0]\}$ of g_0 .

Abstract Execution The above question is answered by an abstract execution of g_0 which mimics the above normal execution of g_0 .

The abstract execution differs from the normal execution in that in place of a substitution is an abstract substitution that describes groundness of the values that variables may take. An abstract substitution associates a mode with a variable. $L \mapsto g$ means that L is a ground term and $H \mapsto u$ means that groundness of H is unknown. Thus, the input abstract substitution of g_0 is $\mu_0 : \{L_1 \mapsto g, L_2 \mapsto g\}$.

The first step of the abstract execution performs an abstract procedure entry. g_0 and μ_0 are first renamed by Ψ . Then C2 is selected and its head $append([H|L_1], L_2, [H|L_3])$ is unified with $\Psi(g_0)$ resulting in E_1 . Then $\Psi(\mu_0) : \{L'_1 \mapsto g, L'_2 \mapsto g\}$ and E_1 are used to compute the input abstract substitution $\mu_1 = \{H \mapsto g, L_1 \mapsto g, L_2 \mapsto g\}$ of g_1 as follows. L'_1 is ground by $\Psi(\mu_0)$ and L'_1 equals to $[H|L_1]$ by E_1 . Therefore, $[H|L_1]$ is ground, which implies that both H and L_1 are ground. So, $H \mapsto g$ and $L_1 \mapsto g$ are in μ_1 . Similarly, $L_2 \mapsto g$ is in μ_1 since $L'_2 \mapsto L_2$ is in E_1 and $L'_2 \mapsto g$ is in $\Psi(\mu_0)$. In this way, the initial goal g_0 with its input abstract substitution μ_0 has been reduced into g_1 and its input abstract substitution μ_1 .

The abstract execution of g_1 , details of which have been omit-

ted, transforms μ_1 into $\mu_2 : \{H \mapsto g, L_1 \mapsto g, L_2 \mapsto g, L_3 \mapsto g\}$ the output abstract substitution of g_1 .

The last step of the abstract execution performs an abstract procedure exit. The head of C2 and μ_2 are renamed by Ψ , and then the renamed head $append([H'|L'_1], L'_2, [H'|L'_3])$ is unified with g_0 resulting in E_2 . $\Psi(\mu_2) : \{H \mapsto g, L_1 \mapsto g, L_2 \mapsto g, L_3 \mapsto g\}$ and E_2 are then used to update the input abstract substitution μ_0 of g_0 and this results in the output abstract substitution $\mu_3 : \{L_1 \mapsto g, L_2 \mapsto g, L_3 \mapsto g\}$. The modes assigned to L_1 and L_2 by μ_3 are the same as those by μ_0 . By $\Psi(\mu_2)$, H' and L' are ground. Hence, $[H'|L']$ is ground. By E_2 , L_3 equals to $[H'|L']$ implying L_3 is ground. So, $L_3 \mapsto g$ is in μ_3 .

C1 may be applied at the first step since μ_0 also describes $\{L_1 \mapsto [], L_2 \mapsto [0, s(0)]\}$. This alternative abstract computation would give the same output abstract substitution $\{L_1 \mapsto g, L_2 \mapsto g, L_3 \mapsto g\}$ of g_0 . Therefore, according to μ_3 , if L_1 and L_2 are ground before $append(L_1, L_2, L_3)$ is executed, L_3 is ground after $append(L_1, L_2, L_3)$ is successfully executed.

The abstract execution closely resembles the normal execution. They differ in that the abstract execution processes abstract substitutions whilst the normal execution processes substitutions and in that the abstract execution performs abstract procedure entries and exits whilst the normal execution performs procedure entries and exits.

Abstract Interpretation Performing program analysis by mimicking normal program execution is called abstract interpretation. The resemblance between normal and abstract executions is common among different kinds of analysis. An abstract interpretation framework factors out common features of normal and abstract executions and models normal and abstract executions by a number of operators on a semantic domain, which leads to the following formalisation of abstract interpretation.

Let \mathbf{C} be a set. An n -ary operator on \mathbf{C} is a function from \mathbf{C}^n to \mathbf{C} . An interpretation \mathbb{C} is a tuple $\langle (\mathbf{C}, \sqsubseteq), (\mathcal{C}_1, \dots, \mathcal{C}_k) \rangle$ where $(\mathbf{C}, \sqsubseteq)$ is a complete lattice and $\mathcal{C}_1, \dots, \mathcal{C}_k$ are operators of fixed arities.

Definition 1. Let

$$\mathbb{C} = \langle (\mathbf{C}, \sqsubseteq), (\mathcal{C}_1, \dots, \mathcal{C}_k) \rangle$$

$$\mathbb{M} = \langle (\mathbf{M}, \preccurlyeq), (\mathcal{M}_1, \dots, \mathcal{M}_k) \rangle$$

be two interpretations such that \mathcal{C}_i and \mathcal{M}_i are of same arity n_i for each $1 \leq i \leq k$, and γ be a monotonic function from \mathbf{M} to \mathbf{C} . \mathbb{M} is called a γ -abstraction of \mathbb{C} if, for each $1 \leq i \leq k$,

– for all $\mathbf{c}_1, \dots, \mathbf{c}_{n_i} \in \mathbf{C}$ and $\mathbf{m}_1, \dots, \mathbf{m}_{n_i} \in \mathbf{M}$,

$$(\wedge_{1 \leq k \leq n_i} \mathbf{c}_k \sqsubseteq \gamma(\mathbf{m}_k)) \rightarrow \mathcal{C}_i(\mathbf{c}_1, \dots, \mathbf{c}_{n_i}) \sqsubseteq \gamma \circ \mathcal{M}_i(\mathbf{m}_1, \dots, \mathbf{m}_{n_i})$$

\mathbb{M} is called an abstract interpretation and \mathbb{C} a concrete interpretation. An object in \mathbb{M} , say \mathbf{M} , is called abstract while an object in \mathbb{C} , say \mathbf{C} , is called concrete. With respect to a given γ , if $\mathbf{c} \sqsubseteq \gamma(\mathbf{m})$ then \mathbf{c} is said to be described by \mathbf{m} or \mathbf{m} is said to be a description of \mathbf{c} . The condition in the above definition is read as that \mathcal{M}_i is a γ -abstraction of \mathcal{C}_i . Therefore, \mathbb{M} is a γ -abstraction of \mathbb{C} iff each operator \mathcal{M} in \mathbb{M} is a γ -abstraction of its corresponding operator \mathcal{C} in \mathbb{C} .

2.3 Abstract Interpretation Framework

The abstract interpretation framework in [27] is based on a collecting semantics of normal logic programs which associates each textual program point with a set of substitutions. The set is a superset of the set of substitutions that may be obtained when the execution of the program reaches that program point. The collecting semantics is defined in terms of two operators on $(\wp(\text{Sub}), \sqsubseteq)$. One operator is the set union \cup - the least upper bound operator on $(\wp(\text{Sub}), \sqsubseteq)$ and the other is *UNIFY* which is defined as follows. Let $a_1, a_2 \in \text{Atom}$ and $\Theta_1, \Theta_2 \in \wp(\text{Sub})$.

$$\text{UNIFY}(a_1, \Theta_1, a_2, \Theta_2) = \{ \text{unify}(a_1, \theta_1, a_2, \theta_2) \neq \text{fail} \mid \theta_1 \in \Theta_1 \wedge \theta_2 \in \Theta_2 \}$$

where

$$\text{unify}(a_1, \theta_1, a_2, \theta_2) \stackrel{\text{def}}{=} \text{mgu}((\Psi(\theta_1))(\Psi(a_1)), \theta_2(a_2)) \circ \theta_2$$

which encompasses both procedure entries and procedure exits. For a procedure entry, a_1 is the calling goal, θ_1 its input substitution, a_2

the head of the selected clause and θ_2 the empty substitution. For a procedure exit, a_1 is the head of the selected clause, θ_1 the output substitution of the last goal of the body of the clause, a_2 the calling goal and θ_2 its input substitution.

The collecting semantics corresponds to the following concrete interpretation.

$$\langle (\wp(Sub), \subseteq), (\cup, UNIFY) \rangle$$

Specialising the framework for a particular program analysis consists in designing an abstract domain $(ASub, \sqsubseteq)$, a concretisation function $\gamma : ASub \mapsto \wp(Sub)$, and two abstract operators \sqcup and $AUNIFY$ such that $\langle (ASub, \sqsubseteq), (\sqcup, AUNIFY) \rangle$ is a γ -abstraction of $\langle (\wp(Sub), \subseteq), (\cup, UNIFY) \rangle$. Once $(ASub, \sqsubseteq)$ and γ are designed, it remains to design $AUNIFY$ such that $AUNIFY$ is a γ -abstraction of $UNIFY$ since \sqcup is a γ -abstraction of \cup . $AUNIFY$ is called an abstract unification operator since its main work is to abstract unification. The abstract unification operator implements both abstract procedure entries and abstract procedure exits as *unify* encompasses both procedure entries and procedure exits. We note that the *procedure-entry* and *procedure-exit* operators in [4], the *type substitution propagation* operator in [22, 23] and the corresponding operators in [35] and [36] can be thought of as variants of $AUNIFY$.

3 Polymorphic abstract interpretation

This section recalls the notion of polymorphic abstract interpretation proposed in [29].

An analysis corresponds to an abstract interpretation. For a monomorphic analysis, data descriptions are monomorphic as they do not contain parameters. For a polymorphic analysis, data descriptions contain parameters. Take the logic program in figure 1 as an example. By a monomorphic type analysis [18, 28], one would infer $[L_1 \in list(nat), L_2 \in list(nat)]append(L_1, L_2, L_3)[L_3 \in list(nat)]$ and $[L_1 \in list(int), L_2 \in list(int)]append(L_1, L_2, L_3)[L_3 \in list(int)]$. It is desirable to design a polymorphic type analysis [3, 8, 29] which would infer $[L_1 \in list(\alpha), L_2 \in list(\alpha)]append(L_1, L_2, L_3)[L_3 \in list(\alpha)]$.

The two statements inferred by the monomorphic type analysis are two instances of the statement inferred by the polymorphic type analysis. A polymorphic analysis is a representation of a possibly

infinite number of monomorphic analyses. The result of a polymorphic analysis subsumes the results of many monomorphic analyses in the sense that the results of the monomorphic analyses may be obtained as instances of the result of the polymorphic analysis. Let $\mathbb{P} = <(\mathbf{P}, \sqsubseteq), (\mathcal{P}_1, \dots, \mathcal{P}_k)>$ be the abstract interpretation for a polymorphic analysis. The elements in \mathbf{P} by necessity contain parameters because the result of the polymorphic analysis contains parameters. Since parameters may take as value any element from an underlying domain, it is necessary to take into account all possible assignments of values to parameters in order to formalise a polymorphic abstraction. In the sequel, an assignment will always mean an assignment of values to the parameters that serve as placeholders for information to be provided after analysis.

Definition 2. Let \mathbf{A} be the set of assignments, $\mathbb{C} = <(\mathbf{C}, \sqsubseteq), (\mathcal{C}_1, \dots, \mathcal{C}_k)>$ and $\mathbb{P} = <(\mathbf{P}, \sqsubseteq), (\mathcal{P}_1, \dots, \mathcal{P}_k)>$ be two interpretations such that \mathcal{C}_i and \mathcal{P}_i are of same arity n_i for each $1 \leq i \leq k$, and $\Upsilon : \mathbf{P} \times \mathbf{A} \mapsto \mathbf{C}$ be monotonic in its first argument. \mathbb{P} is called a polymorphic (Υ, \mathbf{A}) -abstraction of \mathbb{C} if, for each $1 \leq i \leq k$,

- for all $\kappa \in \mathbf{A}$, all $\mathbf{c}_1, \dots, \mathbf{c}_{n_i} \in \mathbf{C}$ and all $\mathbf{p}_1, \dots, \mathbf{p}_{n_i} \in \mathbf{P}$,

$$(\wedge_{1 \leq j \leq n_i} \mathbf{c}_j \sqsubseteq \Upsilon(\mathbf{p}_j, \kappa)) \rightarrow \mathcal{C}_i(\mathbf{c}_1, \dots, \mathbf{c}_{n_i}) \sqsubseteq \Upsilon(\mathcal{P}_i(\mathbf{p}_1, \dots, \mathbf{p}_{n_i}), \kappa)$$

The above condition is read as that \mathcal{P}_i is a polymorphic (Υ, \mathbf{A}) -abstraction of \mathcal{C}_i . Therefore, \mathbb{P} is a polymorphic (Υ, \mathbf{A}) -abstraction of \mathbb{C} iff each operator \mathcal{P} in \mathbb{P} is a polymorphic (Υ, \mathbf{A}) -abstraction of its corresponding operator \mathcal{C} in \mathbb{C} . With \mathbf{A} , Υ and \mathbb{C} being understood, \mathbb{P} is called a polymorphic abstract interpretation. The notion of polymorphic abstract interpretation provides us better understanding of polymorphic analyses and simplifies the design and the proof of polymorphic analyses.

An alternative formulation of a polymorphic abstract interpretation is to define Υ as a function of type $\mathbf{P} \mapsto (\mathbf{A} \mapsto \mathbf{C})$. Thus, a polymorphic abstract interpretation is a special class of abstract interpretation.

4 Monomorphic Groundness Analysis

This section reformulates the groundness analysis presented in [38] that uses the abstract domain for groundness proposed in [32]. The

reformulated groundness analysis will be used section 5 to obtain the polymorphic groundness analysis.

4.1 Abstract Domains

In groundness analysis, we are interested in knowing which variables in \mathcal{V}_P will be definitely instantiated to ground terms and which variables in \mathcal{V}_P are not necessarily instantiated to ground terms. We use g and u to represent these two instantiation modes of a variable. Let $\text{MO} \stackrel{\text{def}}{=} \{g, u\}$ and \sqsubseteq be defined as $g \sqsubseteq g$, $g \sqsubseteq u$ and $u \sqsubseteq u$. $\langle \text{MO}, \sqsubseteq \rangle$ is a complete lattice with infimum g and supremum u . Let \sqcap and \sqcup be the least upper bound and the greatest lower bound operators on $\langle \text{MO}, \sqsubseteq \rangle$ respectively.

The intuition of a mode in MO describing a set of terms is captured by a concretisation function $\gamma_{term} : \langle \text{MO}, \sqsubseteq \rangle \mapsto \langle \wp(\text{Sub}), \subseteq \rangle$ defined in the following.

$$\gamma_{term}(m) \stackrel{\text{def}}{=} \begin{cases} \text{Term}(\Sigma, \emptyset), & \text{if } m = g \\ \text{Term}(\Sigma, \mathcal{V}), & \text{if } m = u \end{cases}$$

A set of substitutions is described naturally by associating each variable in \mathcal{V}_P with a mode from MO . The abstract domain for groundness analysis is thus $\langle \text{MSub}, \sqsubseteq \rangle$ where

$$\text{MSub} \stackrel{\text{def}}{=} \mathcal{V}_P \mapsto \text{MO}$$

and \sqsubseteq is the pointwise extension of \sqsubseteq . $\langle \text{MSub}, \sqsubseteq \rangle$ is a complete lattice. We use \sqcup and \sqcap to denote the least upper bound and the greatest lower bound operators on $\langle \text{MSub}, \sqsubseteq \rangle$ respectively. The set of substitutions described by a function from \mathcal{V}_P to MO is modelled by a concretisation function γ_{sub} from $\langle \text{MSub}, \sqsubseteq \rangle$ to $\langle \wp(\text{Sub}), \subseteq \rangle$ defined as follows.

$$\gamma_{sub}(\theta^b) \stackrel{\text{def}}{=} \{\theta \mid \forall X \in \mathcal{V}_P. (\theta(X) \in \gamma_{term}(\theta^b(X)))\}$$

γ_{sub} is a monotonic function from $\langle \text{MSub}, \sqsubseteq \rangle$ to $\langle \wp(\text{Sub}), \subseteq \rangle$.

Monomorphic abstract substitutions in MSub describes modes of variables in \mathcal{V}_P . The abstract unification operator for the monomorphic groundness analysis will also make use of modes of renamed variables. Let $\mathcal{V}_P^+ \stackrel{\text{def}}{=} \mathcal{V}_P \cup \Psi(\mathcal{V}_P)$. We define $\text{MSub}^+ \stackrel{\text{def}}{=} \mathcal{V}_P^+ \mapsto \text{MO}$ and \sqsubseteq^+ as pointwise extension of \sqsubseteq . γ_{sub}^+ , \sqcup^+ and \sqcap^+ are defined as counterparts of γ_{sub} , \sqcup and \sqcap respectively.

Lemma 3. $\gamma_{term}(\text{MO})$, $\gamma_{sub}(\text{MSub})$ and $\gamma_{sub}^+(\text{MSub}^+)$ are Moore families.

Proof. Straightforward. \square

4.2 Abstract Unification Operator

Algorithm 1 defines an abstract unification operator for groundness analysis. Given $\theta^b, \sigma^b \in \text{MSub}$ and $A, B \in \text{ATOM}(\Sigma, \Pi, \mathcal{V}_P)$, it computes $MUNIFY(A, \theta^b, B, \sigma^b) \in \text{MSub}$ in five steps. In step (1), Ψ is applied to A and θ^b to obtain $\Psi(A)$ and $\Psi(\theta^b)$, and $\Psi(\theta^b)$ and σ^b are combined to obtain $\zeta^b = \Psi(\theta^b) \cup \sigma^b$ so that a substitution satisfying ζ^b satisfies both $\Psi(\theta^b)$ and σ^b . Note that $\zeta^b \in \text{MSub}^+$. In step (2), $E_0 = eq \circ mgu(\Psi(A), B)$ is computed. If $E_0 = \text{fail}$ then the algorithm returns $\{X \mapsto g \mid X \in \mathcal{V}_P\}$ - the infimum of $\langle \text{MSub}, \sqsubseteq \rangle$. Otherwise, the algorithm continues. In step (3), $\eta^b = MDOWN(E_0, \zeta^b)$ is computed so that η^b is satisfied by any $\zeta \circ mgu(E_0 \zeta)$ for any ζ satisfying ζ^b . In step (4), the algorithm computes $\beta^b = MUP(\eta^b, E_0)$ from η^b such that any substitution satisfies β^b if it satisfies η^b and unifies E_0 . In step (5), the algorithm restricts β^b to \mathcal{V}_P and returns the result.

Algorithm 1 Let $\theta^b, \sigma^b \in \text{MSub}$, $A, B \in \text{Atom}(\Sigma, \Pi, \mathcal{V}_P)$.

$$\begin{aligned}
 MUNIFY(A, \theta^b, B, \sigma^b) &\stackrel{\text{def}}{=} \\
 &\begin{cases} \text{let } E_0 = eq \circ mgu(\Psi(A), B) \text{ in} \\ \text{if } E_0 \neq \text{fail} \\ \text{then } MUP(E_0, MDOWN(E_0, \Psi(\theta^b) \cup \sigma^b)) \upharpoonright \mathcal{V}_P \\ \text{else } \{X \mapsto g \mid X \in \mathcal{V}_P\} \end{cases} \\
 MDOWN(E, \zeta^b) &\stackrel{\text{def}}{=} \eta^b \\
 \text{where } \eta^b(X) &\stackrel{\text{def}}{=} \begin{cases} \zeta^b(X), & \text{if } X \notin \text{range}(E) \\ \zeta^b(X) \triangle \Delta_{(Y=t) \in E \wedge X \in \text{vars}(t)} \zeta^b(Y), & \text{otherwise.} \end{cases} \\
 MUP(E, \eta^b) &\stackrel{\text{def}}{=} \beta^b \\
 \text{where } \beta^b(X) &\stackrel{\text{def}}{=} \begin{cases} \eta^b(X), & \text{if } X \notin \text{dom}(E) \\ \eta^b(X) \triangle \bigtriangledown_{Y \in \text{vars}(E(X))} \zeta^b(Y), & \text{otherwise.} \end{cases}
 \end{aligned}$$

The following theorem states that the abstract unification operator is a safe approximation of $UNIFY$.

Theorem 4. For any $\theta^b, \sigma^b \in \text{MSub}$ and any $A, B \in \text{Atom}(\Sigma, \Pi, \mathcal{V}_P)$.

$$\text{UNIFY}(A, \gamma_{\text{sub}}(\theta^b), B, \gamma_{\text{sub}}(\sigma^b)) \subseteq \gamma_{\text{sub}}(\text{MUNIFY}(A, \theta^b, B, \sigma^b))$$

Proof. $\Psi(\theta^b) \cup \sigma^b \in \text{MSub}^+$. Let $\zeta \in \gamma_{\text{sub}}^+(\Psi(\theta^b) \cup \sigma^b)$ and $(Y \mapsto g) \in \text{MDOWN}(E_0, \Psi(\theta^b) \cup \sigma^b)$. Then either $(Y \mapsto g) \in \Psi(\theta^b) \cup \sigma^b$ or there is X and t such that $(X \mapsto g) \in \Psi(\theta^b) \cup \sigma^b$, $(X = t) \in E_0$ and $Y \in \text{vars}(t)$. So, $Y(\zeta \circ \text{mgu}(E_0\zeta))$ is ground if $\text{mgu}(E_0\zeta) \neq \text{fail}$. If every variable in a term is ground under a substitution then that term is ground under the same substitution. Therefore, if $(Z \mapsto g) \in \text{MUP}(E_0, \text{MDOWN}(E_0, \Psi(\theta^b) \cup \sigma^b))$ then Z is ground under $\zeta \circ \text{mgu}(E_0\zeta)$. This completes the proof of the theorem. \square

Example 1. Let $\mathcal{V}_P = \{X, Y, Z\}$, $A = g(X, f(Y, f(Z, Z)), Y)$, $B = g(f(X, Y), Z, X)$, $\theta^b = \{X \mapsto g, Y \mapsto u, Z \mapsto u\}$ and $\sigma^b = \{X \mapsto u, Y \mapsto u, Z \mapsto g\}$. $\text{MUNIFY}(A, \theta^b, B, \sigma^b)$ is computed as follows.

In step (1), $\Psi = \{X \mapsto X_0, Y \mapsto Y_0, Z \mapsto Z_0\}$ is applied to A and θ^b .

$$\begin{aligned}\Psi(A) &= g(X_0, f(Y_0, f(Z_0, Z_0)), Y_0) \\ \Psi(\theta^b) &= \{X_0 \mapsto g, Y_0 \mapsto u, Z_0 \mapsto u\}\end{aligned}$$

and $\zeta^b = \Psi(\theta^b) \cup \sigma^b = \{X_0 \mapsto g, Y_0 \mapsto u, Z_0 \mapsto u, X \mapsto u, Y \mapsto u, Z \mapsto g\}$ is computed.

In step (2), $E_0 = \text{eq} \circ \text{mgu}(\Psi(A), B) = \{X_0 = f(Y_0, Y), Z = f(Y_0, f(Z_0, Z_0)), X = Y_0\}$ is computed.

In step (3), $\eta^b = \text{MDOWN}(E_0, \zeta^b) = \{X_0 \mapsto g, Y_0 \mapsto g, Z_0 \mapsto g, X \mapsto u, Y \mapsto g, Z \mapsto g\}$ is computed.

In step (4), $\beta^b = \text{MUP}(E_0, \eta^b) = \{X_0 \mapsto g, Y_0 \mapsto g, Z_0 \mapsto g, X \mapsto g, Y \mapsto g, Z \mapsto g\}$ is computed.

In step (5), $\beta^b \upharpoonright \mathcal{V}_P = \{X \mapsto g, Y \mapsto g, Z \mapsto g\}$ is returned.

So, $\text{MUNIFY}(A, \theta^b, B, \sigma^b) = \{X \mapsto g, Y \mapsto g, Z \mapsto g\}$. \square

5 Polymorphic Groundness Analysis

We now present the polymorphic groundness analysis. We first design polymorphic domains corresponding to the monomorphic domains for the monomorphic groundness analysis and then obtain the polymorphic groundness analysis by replacing two primitive monomorphic operators by their polymorphic counterparts.

5.1 Abstract Domains

In a polymorphic groundness analysis, the input contains a number Para of mode parameters which may be filled in with values from MO . The set of assignments is hence $\mathbf{A} = \text{Para} \mapsto \text{MO}$.

We first consider how to express mode information in presence of mode parameters. The polymorphic groundness analysis needs to propagate mode parameters in a precise way. The abstract unification operator for the monomorphic groundness analysis computes the least upper bounds and the greatest lower bounds of modes when it propagates mode information. This raises no difficulty in the monomorphic groundness analysis as the two operands of the least upper bound operator or the greatest lower bound operator are modes from MO . In the polymorphic groundness analysis, their operands contains parameters. This makes it clear that mode information can no longer be represented by a single mode value or parameter in order to propagate mode information in a precise manner.

We use as a tentative polymorphic mode description a set of subsets of mode parameters. Thus, the set of tentative polymorphic mode descriptions is $\wp(\wp(\text{Para}))$. The denotation of a set \mathcal{S} of subsets of mode parameters under a given assignment κ is determined as follows. Let $\mathcal{S} = \{S_1, S_2, \dots, S_n\}$ and $S_i = \{\alpha_i^1, \alpha_i^2, \dots, \alpha_i^{k_i}\}$. Then, for a particular assignment κ , \mathcal{S} is interpreted as $\bigtriangledown_{1 \leq i \leq n} \bigtriangleup_{1 \leq j \leq k_i} \kappa(\alpha_i^j)$. \emptyset represents g under any assignment since $\bigtriangledown \emptyset = \text{g}$ while $\{\emptyset\}$ represents u as $\bigtriangledown \bigtriangleup \emptyset = \text{u}$. Let \ll be a relation on $\wp(\wp(\text{Para}))$ defined as follows.

$$\mathcal{S}_1 \ll \mathcal{S}_2 \stackrel{\text{def}}{=} \forall S_1 \in \mathcal{S}_1. \exists S_2 \in \mathcal{S}_2. (S_2 \subseteq S_1)$$

For instance, $\{\{\alpha_1, \alpha_2\}, \{\alpha_1, \alpha_3\}\} \ll \{\{\alpha_1\}\}$ and $\{\{\alpha_1, \alpha_2\}, \{\alpha_1, \alpha_3\}\} \ll \{\{\alpha_2\}, \{\alpha_3\}\}$. If $\mathcal{S}_1 \ll \mathcal{S}_2$ then, under any assignment, \mathcal{S}_1 represents a mode that is smaller than or equal to the mode represented by \mathcal{S}_2 under the same assignment. \ll is a pre-order. It is reflexive and transitive but not antisymmetric. For instance, $\{\{\alpha_1, \alpha_2\}, \{\alpha_1\}\} \ll \{\{\alpha_1\}\}$ and $\{\{\alpha_1\}\} \ll \{\{\alpha_1, \alpha_2\}, \{\alpha_1\}\}$. Define \cong as $\mathcal{S}_1 \cong \mathcal{S}_2 \stackrel{\text{def}}{=} (\mathcal{S}_1 \ll \mathcal{S}_2) \wedge (\mathcal{S}_2 \ll \mathcal{S}_1)$. Then \cong is an equivalence relation on $\wp(\wp(\text{Para}))$. The domain of mode descriptions is constructed as

$$\text{PM} \stackrel{\text{def}}{=} \wp(\wp(\text{Para}))_{/\cong}$$

$$\preceq \stackrel{\text{def}}{=} \ll_{/\cong}$$

$\langle \text{PM}, \preceq \rangle$ is a complete lattice with its infimum being $[\emptyset]_{\cong}$ and its supremum being $[\{\emptyset\}]_{\cong}$. The least upper bound operator \oplus and the greatest lower bound operator \otimes on $\langle \text{PM}, \preceq \rangle$ are given as follows.

$$\begin{aligned} [\mathcal{S}_1]_{\cong} \oplus [\mathcal{S}_2]_{\cong} &\stackrel{\text{def}}{=} [\mathcal{S}_1 \cup \mathcal{S}_2]_{\cong} \\ [\mathcal{S}_1]_{\cong} \otimes [\mathcal{S}_2]_{\cong} &\stackrel{\text{def}}{=} [\{S_1 \cup S_2 \mid S_1 \in \mathcal{S}_1 \wedge S_2 \in \mathcal{S}_2\}]_{\cong} \end{aligned}$$

The meaning of a polymorphic mode description is given by $\Upsilon_{\text{term}} : \text{PM} \times \mathbf{A} \mapsto \wp(\text{Term}(\Sigma, \mathcal{V}))$ defined as follows.

$$\Upsilon_{\text{term}}([\mathcal{S}]_{\cong}, \kappa) \stackrel{\text{def}}{=} \gamma_{\text{term}}(\bigtriangledown_{S \in \mathcal{S}} \bigtriangleup_{s \in S} \kappa(s))$$

Υ_{term} interprets \mathcal{S} as a disjunction of conjunctions of mode parameters. For instance, if a variable X is associated with a mode description $\{\{\alpha_1, \alpha_2\}, \{\alpha_1, \alpha_3\}\}$, then its mode is $(\alpha_1 \Delta \alpha_2) \bigtriangledown (\alpha_1 \Delta \alpha_3)$. For simplicity, a polymorphic mode description will be written as a set of subsets of mode parameters, using a member of an equivalence class of \cong to represent the equivalence class.

Polymorphic abstract substitutions are a function mapping a variable in \mathcal{V}_P to a mode description in PM . Polymorphic abstract substitutions are ordered pointwise. The domain of polymorphic abstract substitutions is $\langle \text{PSub}, \in \rangle$ where $\text{PSub} \stackrel{\text{def}}{=} \mathcal{V}_P \mapsto \text{PM}$ and \in is the pointwise extension of \preceq . $\langle \text{PSub}, \in \rangle$ is a complete lattice with its infimum being $\{X \mapsto \emptyset \mid X \in \mathcal{V}_P\}$ and its supremum being $\{X \mapsto \{\emptyset\} \mid X \in \mathcal{V}_P\}$. The least upper bound operator \uplus and the greatest lower bound operator \sqcap are pointwise extensions of \oplus and \otimes respectively. The meaning of a polymorphic abstract substitution is given by $\Upsilon_{\text{sub}} : \text{PSub} \times \mathbf{A} \mapsto \wp(\text{Sub})$ defined as follows.

$$\Upsilon_{\text{sub}}(\theta^\sharp, \kappa) \stackrel{\text{def}}{=} \{\theta \mid \forall X \in \mathcal{V}_P. (\theta(X) \in \Upsilon_{\text{term}}(\theta^\sharp(X), \kappa))\}$$

We define $\text{PSub}^+ \stackrel{\text{def}}{=} \mathcal{V}_P^+ \mapsto \text{PM}$ and \in^+ as pointwise extension of \preceq . Υ_{sub}^+ , \uplus^+ and \sqcap^+ are defined as counterparts of Υ_{sub} , \uplus and \sqcap respectively.

Lemma 5. $\Upsilon_{\text{term}}(\text{PM})$, $\Upsilon_{\text{sub}}(\text{PSub})$ and $\Upsilon_{\text{sub}}^+(\text{PSub}^+)$ are Moore families.

Proof. For any $\kappa \in \mathbf{A}$, we have $\Upsilon_{term}(\{\emptyset\}) = \mathbf{Sub}$, implying that Υ_{term} is co-strict. Let $\mathcal{S}_1, \mathcal{S}_2 \in \mathbf{PM}$ and $\kappa \in \mathbf{A}$. Then either (a) $\Upsilon_{term}(\mathcal{S}_1 \otimes \mathcal{S}_2, \kappa) = \gamma_{term}(\mathbf{g})$ or (b) $\Upsilon_{term}(\mathcal{S}_1 \otimes \mathcal{S}_2, \kappa) = \gamma_{term}(\mathbf{u})$. In the case (a), we have that $\Delta_{\alpha \in \mathcal{S}_1} \kappa(\alpha)$ for any $S_1 \in \mathcal{S}_1$ and $\Delta_{\beta \in \mathcal{S}_2} \kappa(\beta)$ for any $S_2 \in \mathcal{S}_2$. This implies that $\Upsilon_{term}(\mathcal{S}_1, \kappa) = \gamma_{term}(\mathbf{g})$ and $\Upsilon_{term}(\mathcal{S}_2, \kappa) = \gamma_{term}(\mathbf{g})$. Therefore, $\Upsilon_{term}(\mathcal{S}_1 \otimes \mathcal{S}_2, \kappa) = \Upsilon_{term}(\mathcal{S}_1, \kappa) \cap \Upsilon_{term}(\mathcal{S}_2, \kappa)$ holds in the case (a). That $\Upsilon_{term}(\mathcal{S}_1 \otimes \mathcal{S}_2, \kappa) = \Upsilon_{term}(\mathcal{S}_1, \kappa) \cap \Upsilon_{term}(\mathcal{S}_2, \kappa)$ holds in the case (b) can be proven similarly. So, $\Upsilon_{term}(\mathbf{PM})$ is a Moore family.

That $\Upsilon_{sub}(\mathbf{PSub})$ and $\Upsilon_{sub}^+(\mathbf{PSub}^+)$ are Moore families is an immediate consequence of that $\Upsilon_{term}(\mathbf{PM})$ is a Moore family.

5.2 Abstract Unification Operator

Algorithm 2 defines an abstract unification operator for the polymorphic groundness analysis. It is obtained from that for the monomorphic groundness analysis by replacing monomorphic descriptions with polymorphic descriptions, ∇ and Δ by \oplus and \otimes respectively, and renaming *MUNIFY*, *MDOWN* and *MUP* into *PUNIFY*, *PDOWN* and *PUP* respectively.

Algorithm 2 Let $\theta^\sharp, \sigma^\sharp \in \mathbf{PSub}$, $A, B \in \mathbf{Atom}(\Sigma, \Pi, \mathcal{V}_P)$.

$$\begin{aligned}
 PUNIFY(A, \theta^\sharp, B, \sigma^\sharp) &\stackrel{\text{def}}{=} \\
 &\begin{cases} \text{let } E_0 = eq \circ mgu(\Psi(A), B) \text{ in} \\ \text{if } E_0 \neq \text{fail} \\ \text{then } PUP(E_0, PDOWN(E_0, \Psi(\theta^\sharp) \cup \sigma^\sharp)) \upharpoonright \mathcal{V}_P \\ \text{else } \{X \mapsto \emptyset \mid X \in \mathcal{V}_P\} \end{cases} \\
 PDOWN(E, \zeta^\sharp) &\stackrel{\text{def}}{=} \eta^\sharp \\
 \text{where } \eta^\sharp(X) &\stackrel{\text{def}}{=} \begin{cases} \zeta^\sharp(X), & \text{if } X \notin \text{range}(E) \\ \zeta^\sharp(X) \otimes \otimes_{(Y=t) \in E \wedge X \in \text{vars}(t)} \zeta^\sharp(Y), & \text{otherwise.} \end{cases} \\
 PUP(E, \eta^\sharp) &\stackrel{\text{def}}{=} \beta^\sharp \\
 \text{where } \beta^\sharp(X) &\stackrel{\text{def}}{=} \begin{cases} \eta^\sharp(X), & \text{if } X \notin \text{dom}(E) \\ \eta^\sharp(X) \otimes \oplus_{Y \in \text{vars}(E(X))} \zeta^\sharp(Y), & \text{otherwise.} \end{cases}
 \end{aligned}$$

Example 2. Let

$$\begin{aligned}\mathcal{V}_P &= \{X, Y, Z\} \\ A &= g(X, f(Y, f(Z, Z)), Y) \\ B &= g(f(X, Y), Z, X) \\ \theta^\sharp &= \{X \mapsto \{\{\alpha_1, \alpha_2\}\}, Y \mapsto \{\{\alpha_1, \alpha_3\}\}, Z \mapsto \{\{\alpha_2, \alpha_3\}\}\} \\ \sigma^\sharp &= \{X \mapsto \{\{\alpha_1\}, \{\alpha_2\}\}, Y \mapsto \{\{\alpha_2, \alpha_3\}\}, Z \mapsto \{\emptyset\}\}\end{aligned}$$

$PUNIFY(A, \theta^\sharp, B, \sigma^\sharp)$ is computed as follows.

In step (1), $\Psi = \{X \mapsto X_0, Y \mapsto Y_0, Z \mapsto Z_0\}$ is applied to A and θ^\sharp .

$$\begin{aligned}\Psi(A) &= g(X_0, f(Y_0, f(Z_0, Z_0)), Y_0) \\ \Psi(\theta^\sharp) &= \{X_0 \mapsto \{\{\alpha_1, \alpha_2\}\}, Y_0 \mapsto \{\{\alpha_1, \alpha_3\}\}, Z_0 \mapsto \{\{\alpha_2, \alpha_3\}\}\}\end{aligned}$$

and $\zeta^\sharp = \Psi(\theta^\sharp) \cup \sigma^\sharp = \{X_0 \mapsto \{\{\alpha_1, \alpha_2\}\}, Y_0 \mapsto \{\{\alpha_1, \alpha_3\}\}, Z_0 \mapsto \{\{\alpha_2, \alpha_3\}\}, X \mapsto \{\{\alpha_1\}, \{\alpha_2\}\}, Y \mapsto \{\{\alpha_2, \alpha_3\}\}, Z \mapsto \{\emptyset\}\}$ is computed.

In step (2), $E_0 = eq \circ mgu(\Psi(A), B) = \{X_0 = f(Y_0, Y), Z = f(Y_0, f(Z_0, Z_0)), X = Y_0\}$ is computed.

Step (3) computes

$$\begin{aligned}\eta^\sharp &= PDOWN(E_0, \zeta^\sharp) \\ &= \left\{ \begin{array}{l} X_0 \mapsto \{\{\alpha_1, \alpha_2\}\}, Y_0 \mapsto \{\{\alpha_1, \alpha_2, \alpha_3\}\}, Z_0 \mapsto \{\{\alpha_2, \alpha_3\}\}, \\ X \mapsto \{\{\alpha_1\}, \{\alpha_2\}\}, Y \mapsto \{\{\alpha_1, \alpha_2, \alpha_3\}\}, Z \mapsto \{\emptyset\} \end{array} \right\}\end{aligned}$$

Step (4) computes

$$\begin{aligned}\beta^\sharp &= PUP(E_0, \eta^\sharp) \\ &= \left\{ \begin{array}{l} X_0 \mapsto \{\{\alpha_1, \alpha_2, \alpha_3\}\}, Y_0 \mapsto \{\{\alpha_1, \alpha_2, \alpha_3\}\}, Z_0 \mapsto \{\{\alpha_2, \alpha_3\}\}, \\ X \mapsto \{\{\alpha_1, \alpha_2, \alpha_3\}\}, Y \mapsto \{\{\alpha_1, \alpha_2, \alpha_3\}\}, Z \mapsto \{\{\alpha_2, \alpha_3\}\} \end{array} \right\}\end{aligned}$$

In step (5), $\beta^\sharp \upharpoonright \mathcal{V}_P = \{X \mapsto \{\{\alpha_1, \alpha_2, \alpha_3\}\}, Y \mapsto \{\{\alpha_1, \alpha_2, \alpha_3\}\}, Z \mapsto \{\{\alpha_2, \alpha_3\}\}\}$ is returned. So, $PUNIFY(A, \theta^\sharp, B, \sigma^\sharp) = \{X \mapsto \{\{\alpha_1, \alpha_2, \alpha_3\}\}, Y \mapsto \{\{\alpha_1, \alpha_2, \alpha_3\}\}, Z \mapsto \{\{\alpha_2, \alpha_3\}\}\}$. \square

We now prove that for any assignment, the instantiated result of the polymorphic groundness analysis is as precise as that of the monomorphic groundness analysis corresponding to the assignment.

Theorem 6. Let $\mathcal{S}_1, \mathcal{S}_1 \in \mathbf{PM}$, $\mathbf{m}_1, \mathbf{m}_2 \in \mathbf{MO}$, $\theta^\sharp, \sigma^\sharp \in \mathbf{PSub}$, $\eta^\sharp, \zeta^\sharp \in \mathbf{PSub}^+$, $\theta^\flat, \sigma^\flat \in \mathbf{MSub}$, $\eta^\flat, \zeta^\flat \in \mathbf{MSub}^+$, $A, B \in \mathbf{Atom}(\Sigma, \Pi, \mathcal{V}_P)$ and $E \in \wp(\mathbf{Eqn})$. For any $\kappa \in \mathbf{A}$,

(a) if $\Upsilon_{term}(\mathcal{S}_1, \kappa) = \gamma_{term}(\mathbf{m}_1)$ and $\Upsilon_{term}(\mathcal{S}_2, \kappa) = \gamma_{term}(\mathbf{m}_2)$ then both $\Upsilon_{term}(\mathcal{S}_1 \otimes (\mathcal{S}_2, \kappa) = \gamma_{term}(\mathbf{m}_1 \triangle \mathbf{m}_2)$ and $\Upsilon_{term}(\mathcal{S}_1 \oplus (\mathcal{S}_2, \kappa) = \gamma_{term}(\mathbf{m}_1 \triangleright \mathbf{m}_2)$;

(b) if $\Upsilon_{sub}^+(\zeta^\sharp, \kappa) = \gamma_{sub}^+(\zeta^\flat)$ then

$$\Upsilon_{sub}^+(PDOWN(E, \zeta^\sharp), \kappa) = \gamma_{sub}^+(MDOWN(E, \zeta^\flat))$$

(c) if $\Upsilon_{sub}^+(\eta^\sharp, \kappa) = \gamma_{sub}^+(\eta^\flat)$ then

$$\Upsilon_{sub}^+(PUP(E, \eta^\sharp), \kappa) = \gamma_{sub}^+(MUP(E, \eta^\flat))$$

and

(d) if $\Upsilon_{sub}(\theta^\sharp, \kappa) = \gamma_{sub}(\theta^\flat)$ and $\Upsilon_{sub}(\sigma^\sharp, \kappa) = \gamma_{sub}(\sigma^\flat)$ then

$$\Upsilon_{sub}(PUNIFY(A, \theta^\sharp, B, \sigma^\sharp), \kappa) = \gamma_{sub}(MUNIFY(A, \theta^\flat, B, \sigma^\flat))$$

Proof. (b) and (c) follow from (a) and they together imply (d). Thus, it remains to prove (a).

Assume $\Upsilon_{term}(\mathcal{S}_1, \kappa) = \gamma_{term}(\mathbf{m}_1)$ and $\Upsilon_{term}(\mathcal{S}_2, \kappa) = \gamma_{term}(\mathbf{m}_2)$. The proof of $\Upsilon_{term}(\mathcal{S}_1 \otimes (\mathcal{S}_2, \kappa) = \gamma_{term}(\mathbf{m}_1 \triangle \mathbf{m}_2)$ is done by considering four different combinations of values that \mathbf{m}_1 and \mathbf{m}_2 can take. We only prove it for the case $\mathbf{m}_1 = \mathbf{g}$ and $\mathbf{m}_2 = \mathbf{u}$ as other cases are similar. $\mathbf{m}_1 = \mathbf{g}$ implies $\Delta_{\alpha \in S_1} \kappa(\alpha) = \mathbf{g}$ for any $S_1 \in \mathcal{S}_1$. Thus $\Delta_{\alpha \in S_1 \cup S_2} \kappa(\alpha) = \mathbf{g}$ for any $S_1 \in \mathcal{S}_1$ and $S_2 \in \mathcal{S}_2$. So, $\Upsilon_{term}(\mathcal{S}_1 \otimes (\mathcal{S}_2, \kappa) = \gamma_{term}(\mathbf{m}_1 \triangle \mathbf{m}_2)$ as $\mathbf{m}_1 \triangle \mathbf{m}_2 = \mathbf{g}$.

$\Upsilon_{term}(\mathcal{S}_1 \oplus (\mathcal{S}_2, \kappa) = \gamma_{term}(\mathbf{m}_1 \triangleright \mathbf{m}_2)$ can be proven similarly. \square

The following theorem states that the abstract unification operator $PUNIFY$ for the polymorphic groundness analysis is a safe approximation of $UNIFY$.

Theorem 7. For any $A, B \in \text{Atom}(\Sigma, \Pi, \mathcal{V}_P)$, $\kappa \in \mathbf{A}$, and $\theta^\sharp, \sigma^\sharp \in \text{PSub}$,

$$UNIFY(A, \Upsilon_{sub}(\theta^\sharp, \kappa), B, \Upsilon_{sub}(\sigma^\sharp, \kappa)) \subseteq \Upsilon_{sub}(PUNIFY(A, \theta^\sharp, B, \sigma^\sharp), \kappa)$$

Proof. Let θ^\flat be such that $\gamma_{sub}(\theta^\flat) = \Upsilon_{sub}(\theta^\sharp, \kappa)$ and σ^\flat be such that $\gamma_{sub}(\sigma^\flat) = \Upsilon_{sub}(\sigma^\sharp, \kappa)$. By theorem 4, we have

$$UNIFY(A, \Upsilon_{sub}(\theta^\sharp, \kappa), B, \Upsilon_{sub}(\sigma^\sharp, \kappa)) \subseteq \gamma_{sub}(MUNIFY(A, \theta^\flat, B, \sigma^\flat))$$

By theorem 6, we have

$$\Upsilon_{sub}(PUNIFY(A, \theta^\sharp, B, \sigma^\sharp), \kappa) = \gamma_{sub}(MUNIFY(A, \theta^\flat, B, \sigma^\flat))$$

Therefore,

$$UNIFY(A, \Upsilon_{sub}(\theta^\sharp, \kappa), B, \Upsilon_{sub}(\sigma^\sharp, \kappa)) \subseteq \Upsilon_{sub}(PUNIFY(A, \theta^\sharp, B, \sigma^\sharp), \kappa)$$

This completes the proof of the theorem. \square

6 Implementation and Examples

We have implemented the abstract interpretation framework and the polymorphic groundness analysis in SWI-Prolog. The abstract interpretation framework is implemented using O’Keefe’s least fixed-point algorithm [37]. Both the abstract interpretation framework and the polymorphic type analysis are implemented as meta-interpreters using ground representations for program variables and mode parameters.

6.1 Examples

The following examples present the results of the polymorphic groundness analysis on some Prolog programs. The sets are represented by lists. $V \mapsto T$ is written as V/T in the results, α as *alpha*, and β as *beta*.

Example 3. The following is the Prolog program from [40] (p. 250) for looking up the value for a given key in a dictionary represented as a binary tree and the result of the polymorphic groundness analysis.

```

:-      %[K/[[alpha]],D/[[beta]],V/[[gamma]]]
       lookup(K,D,V)
       %[K/[[alpha,beta]],D/[[beta]],V/[[beta,gamma]]].
```

```

lookup(K,dict(K,X,L,R),V) :-
       %[K/[[alpha,beta]],X/[[beta]],L/[[beta]],R/[[beta]],V/[[gamma]]],
       X = V,
       %[K/[[alpha,beta]],X/[[beta,gamma]],L/[[beta]],R/[[beta]],
       % V/[[beta,gamma]]].
lookup(K,dict(K1,X,L,R),V) :-
       %[K/[[alpha]],K1/[[beta]],X/[[beta]],L/[[beta]],R/[[beta]],
       % V/[[gamma]]],
       K < K1,
       %[K/[],K1/[],X/[[beta]],L/[[beta]],R/[[beta]],V/[[gamma]]],
       lookup(K,L,V),
       %[K/[],K1/[],X/[[beta]],L/[[beta]],R/[[beta]],V/[[beta,gamma]]].
lookup(K,dict(K1,X,L,R),V) :-
```

```

%[K/[[alpha]],K1/[[beta]],X/[[beta]],L/[[beta]],R/[[beta]],
  V/[[gamma]]],
K > K1,
  %[K/[],K1/[],X/[[beta]],L/[[beta]],R/[[beta]],V/[[gamma]]],
  lookup(K,R,V),
  %[K/[],K1/[],X/[[beta]],L/[[beta]],R/[[beta]],V/[[beta,gamma]]].

```

The analysis is done for the goal $lookup(K, D, V)$ and the input abstract substitution $\{K \mapsto \{\{\alpha\}\}, D \mapsto \{\{\beta\}\}, V \mapsto \{\{\gamma\}\}\}$. The input abstract substitution states that before the goal is executed, the instantiation mode of K is α , that of D is β and that of V is γ .

The analysis result indicates that, after the goal is executed, the instantiation mode of K is $\alpha \Delta \beta$, that of D remains β and that of V becomes $\beta \Delta \gamma$. This is captured by the output abstract substitution.

The result can be instantiated by one of eight assignments in $\{\alpha, \beta, \gamma\} \mapsto \text{MO}$. Under a given assignment, a polymorphic mode description evaluates to a mode in $\{g, u\}$. Let $\kappa = \{\alpha \mapsto g, \beta \mapsto g, \gamma \mapsto u\}$. Under the assignment κ , the input abstraction substitution evaluates to $\{K \mapsto g, D \mapsto g, V \mapsto u\}$ and output abstract substitution evaluates to $\{K \mapsto g, D \mapsto g, V \mapsto g\}$. This indicates that if the goal $lookup(K, D, V)$ called with K and D being ground then V is ground after $lookup(K, D, V)$ is successfully executed.

The result also indicates that immediately before the built-in calls $K < K1$ and $K > K1$ are executed, the instantiation modes of K and $K1$ are respectively α and β . The Prolog requires that both operands of the built-in predicates " $<$ " and " $>$ " are ground before their invocations. It is obvious that if α and β are assigned g then the requirement is satisfied and corresponding run-time checks can be eliminated. Such inference of sufficient conditions on inputs for safely removing run-time checks is enabled by the ability of propagating parameters. Without propagation of parameters, the inference would require a backward analysis. Current frameworks for abstract interpretation of logic programs do not support backward analyses.

□

Example 4. The following is the result of polymorphic groundness analysis of the permutation sorting program from [40] (p. 55). The analysis is done with the goal $sort(Xs, Ys)$ and the input abstract substitution $\{Xs \mapsto \{\{\alpha\}\}, Ys \mapsto \{\{\beta\}\}\}$. The output abstract substitution obtained is $\{Xs \mapsto \{\{\alpha\}\}, Ys \mapsto \{\{\alpha, \beta\}\}\}$. This indicates

that the instantiation mode of Ys after the successful execution of $sort(Xs, Ys)$ is the greatest lower bound of the instantiation modes of Xs and Ys prior to the execution of the goal.

The abstract substitution associated with the program point immediately before the built-in call $X =< Y$ associates both X and Y with the polymorphic mode description $\{\{\alpha, \beta\}\}$. For this mode description to evaluate to g , it is sufficient to assign g to either α or β .

```

:-      %[Xs/[[alpha]],Ys/[[beta]]]
sort(Xs,Ys)
      %[Xs/[[alpha]],Ys/[[alpha,beta]]].
```

```

select(X,[X|Xs],Xs) :-
      %[X/[[alpha,beta]],Xs/[[alpha]]].
```

```

select(X,[Y|Ys],[Y|Zs]) :-
      %[X/[[beta]],Y/[[alpha]],Ys/[[alpha]],Zs/[]],
      select(X,Ys,Zs),
      %[X/[[alpha,beta]],Y/[[alpha]],Ys/[[alpha]],Zs/[[alpha]]].
```

```

ordered([]) :-
      %[[]].
```

```

ordered([X]) :-
      %[X/[[alpha,beta]]].
```

```

ordered([X,Y|Ys]) :-
      %[X/[[alpha,beta]],Y/[[alpha,beta]],Ys/[[alpha,beta]]],
      X =< Y,
      %[X/[],Y/[],Ys/[[alpha,beta]]],
      ordered([Y|Ys]),
      %[X/[],Y/[],Ys/[]].
```

```

permutation(Xs,[Z|Zs]) :-
      %[Xs/[[alpha]],Z/[[beta]],Zs/[[beta]],Ys/[]],
      select(Z,Xs,Ys),
      %[Xs/[[alpha]],Z/[[alpha,beta]],Zs/[[beta]],Ys/[[alpha]]],
      permutation(Ys,Zs),
      %[Xs/[[alpha]],Z/[[alpha,beta]],Zs/[[alpha,beta]],Ys/[[alpha]]].
```

```

permutation([],[]) :-
      %[[]].
```

```

sort(Xs,Ys) :-
      %[Xs/[[alpha]],Ys/[[beta]]],
      permutation(Xs,Ys),
      %[Xs/[[alpha]],Ys/[[alpha,beta]]],
      ordered(Ys),
      %[Xs/[[alpha]],Ys/[[alpha,beta]]].
```

□

Example 5. The following is the pure *factorials* program from [40] (p. 39) together with the result of the polymorphic groundness analysis. The program is analysed with the goal $\text{factorial}(N, F)$ with input abstract substitution being $\{N \mapsto \{\{\alpha\}\}, F \mapsto \{\{\beta\}\}\}$. The result shows that the goal $\text{factorial}(N, F)$ always succeeds with N and F being bound to ground terms regardless of the instantiation modes of N and F before the goal $\text{factorial}(N, F)$ is executed.

```

:-      %[N/[[alpha]],F/[[beta]]]
factorial(N,F)
  %[N/[],F/[]].

natural_number(0) :-
  %[[]].
natural_number(s(X)) :-
  %[X/[]],
  natural_number(X),
  %[X/[]].

plus(0,X,X) :-
  %[X/[]],
  natural_number(X),
  %[X/[]].
plus(s(X),Y,s(Z)) :-
  %[X/[],Y/[],Z/[]],
  plus(X,Y,Z),
  %[X/[],Y/[],Z/[]].

times(0,X,0) :-
  %[X/[]].
times(s(X),Y,Z) :-
  %[X/[],Y/[],Z/[],XY/[]],
  times(X,Y,XY),
  %[X/[],Y/[],Z/[],XY/[]],
  plus(XY,Y,Z),
  %[X/[],Y/[],Z/[],XY/[]].

factorial(0,s(0)) :-
  %[[]].
factorial(s(N),F) :-
  %[N/[[alpha]],F/[],F1/[]],
  factorial(N,F1),
  %[N/[],F/[],F1/[]],
  times(s(N),F1,F),
  %[N/[],F/[],F1/[]].

```

□

6.2 Performance

The SWI-Prolog implementation of the polymorphic groundness analysis has been tested with a set of benchmark programs that have been used to evaluate program analyses of logic programs. The experiments are done on a 5x86 IBM compatible PC running Windows95.

The table in figure 2 illustrates the time performance of the polymorphic groundness analysis. Every but the last row corresponds to the result of the polymorphic groundness analysis of a specific input. The input consists of a program, a goal and an input abstract substitution that specifies the modes of the variables in the goal. The program and the goal are listed in the first and the third column. The input abstract substitution is the most general for the goal that associates each variable in the goal with a different mode parameter. For instance, the abstract substitution for the first row is $\{X \mapsto \{\{\alpha\}\}, Y \mapsto \{\{\beta\}\}\}$. The second column lists the size of the program. The fourth column is the time in seconds spent on the polymorphic groundness analysis of the input. Last row gives the total size of the programs and the total time.

The table indicates that the prototype polymorphic groundness analyzer spends an average of 0.0443 seconds to process one line program. This is an acceptable speed for many logic program programs. Both the abstract interpretation framework and the polymorphic groundness analysis are implemented as meta-interpreters in a public domain Prolog system. Moreover, we use ground representations for program variables and mode parameters. Using ground representations enables us to make the prototype implementation in a short time and to avoid possible difficulties that may arise due to improper use of meta-level and object-level variables. However, it also prevents us from taking advantages of built-in unification and forces us to code unification in Prolog. We believe that both the use of meta-programming and that of ground representations significantly slow the prototype. Therefore, there is much space for improving the time performance of the polymorphic groundness analysis through a better implementation.

The same table compares the performance of the polymorphic groundness analysis with that of the monomorphic groundness analysis presented in [38]. The monomorphic groundness analysis in [38]

Program	Lines	Goal	Poly (sec)	Mono (sec)	Ratio	Assignments
graph connectivity	32	connected(X,Y)	0.16	0.097	1.641	4
merge	15	merge(Xs,Ys,Zs)	0.28	0.117	2.382	8
buggy naive reverse	12	nrev(X,Y)	0.11	0.067	1.629	4
buggy quick sort	33	qs(Li,Lo)	0.38	0.182	2.082	4
improved quick sort	20	iqsort(Xs,Ys)	0.33	0.112	2.933	4
tree sort	31	treesort(Xs,Ys)	0.44	0.122	3.591	4
list difference	14	diff(X,Y,Z)	0.11	0.040	2.750	8
list insertion	19	insert(X,Y,Z)	0.11	0.068	1.600	8
quicksort with difference list	20	quicksort(Xs,Ys)	0.28	0.110	2.545	4
dictionary lookup in binary trees	12	lookup(K,D,V)	0.17	0.062	2.720	8
permutation sort	22	sort(Xs,Ys)	0.05	0.057	0.869	4
heapify binary trees	25	heapify(Tree,Heap)	0.55	0.192	2.857	4
exponentiation by multiplication	23	exp(N,X,Y)	0.16	0.078	2.031	8
factorial	22	factorial(N,F)	0.06	0.067	0.888	4
zebra	50	zebra(E,S,J,U,N,Z,W)	0.66	0.325	2.026	128
tsp	148	tsp(N,V,M,S,C)	2.74	1.458	1.878	32
chat	1040	chart_parser	61.08	61.410	0.994	1
neural	381	test(X,N)	6.86	5.510	1.243	4
disj_r	164	top(K)	1.82	1.505	1.209	2
dnf	84	go	3.68	2.960	1.243	1
rev	12	rev(Xs,Ys)	0.11	0.027	4.000	4
tree order	34	v2t(X,Y,Z)	0.93	0.225	4.133	8
serialize	45	go(S)	0.88	0.305	2.885	2
cs_r	314	pgenconfig(C)	12.53	9.910	1.264	2
kalah	272	play(G,R)	4.01	3.212	1.248	4
press	370	test_press(X,Y)	28.94	15.325	1.888	4
queens	29	queens(X,Y)	0.16	0.097	1.641	4
read	437	read(X,Y)	35.09	26.710	1.313	4
rotate	15	rotate(X,Y)	0.22	0.040	5.500	4
ronp	101	puzzle(X)	2.20	1.015	2.167	2
small	11	select_cities(W,C1,C2,C3)	0.06	0.016	3.555	16
peep	417	compeepopt(Pi,O,Pr)	23.94	12.090	1.980	8
gabriel	111	main(V1,V2)	2.14	1.112	1.923	4
naughts-and-crosses	123	play(R)	1.15	0.880	1.306	2
semi	184	go(N,T)	13.68	6.892	1.984	4
	4642		206.07		2.168	9

Fig. 2. Performance of Polymorphic Groundness Analysis

uses a subset of \mathcal{V}_P as an abstract substitution. The subset of \mathcal{V}_P contains those variables that are definitely ground under all concrete substitutions described by the abstract substitution. This allows operators on abstract substitutions to be optimised. The monomorphic groundness analysis is implemented in the same way as the polymorphic groundness analysis.

The number of different assignments to the mode parameters in the input abstract substitution for the polymorphic groundness analysis is two to the power of the number of the mode parameters. Each assignment corresponds to a monomorphic groundness analysis that is performed and measured. The fifth column lists the average time in seconds spent on these monomorphic groundness analyses. The sixth column lists the ratio of the fourth column by the fifth column. It is the ratio of the performance of the polymorphic groundness analysis by that of the monomorphic groundness analysis. The seventh column lists the number of the possible assignments for the mode parameters.

The table shows that the time the polymorphic groundness analysis takes is from 0.869 to 5.500 times that the monomorphic groundness analysis takes on the suit of programs. In average, the polymorphic groundness analysis is 2.168 times slower. This is due to the fact that the polymorphic mode descriptions are more complex than the monomorphic mode descriptions. The abstract unification operator and the least upper bound operator for the polymorphic groundness are more costly than those for the monomorphic groundness analysis.

The result of the polymorphic groundness analysis is much more general than that of the monomorphic groundness analysis. It can be instantiated as many times as there are different assignments for the mode parameters in the input abstract substitution. The average number of different assignments is 9 which is 4.151 times the average performance ratio. This indicates that if all different monomorphic groundness analyses corresponding to a polymorphic groundness analysis are required, polymorphic groundness analysis is 4.151 times better. Moreover, the seven rows with three mode parameters have an average performance ratio of 2.513 and 8 assignments. For these seven rows, the polymorphic groundness analysis is 3.182 times better if all different monomorphic groundness analyses corresponding to a polymorphic groundness analysis are required. Furthermore,

the three rows with four or more mode parameters have an average performance ratio of 2.486 and an average of 58.666 different assignments. Polymorphic groundness analysis is 23.598 times better for these three rows if all different monomorphic groundness analyses are required.

7 Conclusion

We have presented a new groundness analysis, called polymorphic groundness analysis, that infers dependency of the groundness of the variables at a program point on mode parameters that are input to the groundness analysis and can be instantiated after analysis. The polymorphic groundness analysis is obtained by simulating a monomorphic groundness analysis. Some experimental results with a prototype implementation of the analysis are promising. The polymorphic groundness analysis is proved to be as precise as the monomorphic groundness analysis.

The monomorphic groundness analysis that we consider in this paper is the least powerful one. It uses a simple domain for groundness. As groundness is useful both in compile-time program optimisations itself and in improving precisions of other program analyses such as sharing [38, 10, 34, 41, 20], more powerful domains for groundness have been studied. These domains consist of propositional formulae over program variables that act as propositional variables. Dart uses the domain **Def** of definite propositional formulae to capture groundness dependency between variables [15]. For instance, the definite propositional formula $x \leftrightarrow (y \wedge z)$ represents the groundness dependency that x is bound to a ground term if and only if y and z are bound to ground terms. **Def** consists of propositional formulae whose models are closed under set intersection [11]. Marriott and Søndergaard use the domain **Pos** of positive propositional formulae [30]. A positive propositional formula is true whenever each propositional variable it contains is true. **Pos** is strictly more powerful than **Def**. It has been further studied in [11, 1, 2] and has several implementations [5, 25, 17, 7, 9].

The domain for the monomorphic groundness analysis in this paper is isomorphic to a subdomain **Con** of **Pos**. **Con** consists of propositional formulae that are conjunctions of propositional variables.

The polymorphic groundness analysis infers the dependencies of the groundness of variables of interest at a program point on mode parameters while a **Pos**-based groundness analysis infers groundness dependencies among variables of interest at a program point. **Pos**-based groundness analysis can also be used to infer groundness dependencies between variables at a program point and variables in the goal. But this requires a complex program transformation that introduces a new predicate for each program point of interest and may lead to solving large boolean equations involving many propositional variables. Also, the less powerful domain **Def** cannot be used to infer groundness dependencies between variables at a program point and variables in the goal because, unlike **Pos**, **Def** is not condensing [20, 30].

Though the polymorphic groundness analysis is not intended for inferring groundness dependencies among variables of interest at a program point, it captures this kind of dependency indirectly. In example 4, the output abstract substitution for the goal $sort(Xs, Ys)$ is $\{Xs \mapsto \{\{\alpha\}\}, Ys \mapsto \{\{\alpha, \beta\}\}\}$. This implies that whenever Xs is bound to a ground term, Ys is bound to a ground term because assigning g to α will evaluate $\{\{\alpha, \beta\}\}$ to g regardless of the mode assigned to β . In general, if the abstract substitution at a program point assign \mathcal{R}_j to Y_j for $1 \leq j \leq l$ and \mathcal{S}_i to X_i for $1 \leq i \leq k$ and $\oplus_{1 \leq j \leq l} \mathcal{R}_j \preceq \oplus_{1 \leq i \leq k} \mathcal{S}_k$ then the **Pos** like proposition $\wedge_{1 \leq i \leq k} X_i \rightarrow \wedge_{1 \leq j \leq l} Y_j$ holds at the program point. Thus the polymorphic groundness analysis can also infer the groundness dependencies produced by a **Pos**-based groundness analysis. Moreover, it can be directly plugged into most abstract interpretation frameworks for logic programs. However, this kind of groundness dependency in the result of the polymorphic groundness analysis is not as explicit as in the result of a **Pos**-based analysis.

References

- [1] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Boolean functions for dependency analysis: Algebraic properties and efficient representation. In B. Le Charlier, editor, *Static Analysis: Proceedings of the First International Symposium*, Lecture Notes in Computer Science 864, pages 266–280. Springer-Verlag, 1994.
- [2] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two classes of Boolean functions for dependency analysis. *Science of Computer Programming*, 31(1):3–45, 1998.

- [3] R. Barbuti and R. Giacobazzi. A bottom-up polymorphic type inference in logic programming. *Science of computer programming*, 19(3):133–181, 1992.
- [4] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming*, 10(2):91–124, 1991.
- [5] Baudouin Le Charlier and Pascal Van Hentenryck. Groundness analysis for prolog: Implementation and evaluation of the domain prop. Technical Report CS-92-49, Department of Computer Science, Brown University, October 1992. Sun, 13 Jul 1997 18:30:16 GMT.
- [6] M. Codish, D. Dams, and Yardeni E. Derivation and safety of an abstract unification algorithm for groundness and aliasing analysis. In Furukawa [16], pages 79–93.
- [7] M. Codish and B. Demoen. Analysing logic programs using “Prop”-ositional logic programs and a magic wand. In Dale Miller, editor, *Logic Programming - Proceedings of the 1993 International Symposium*, pages 114–129, Massachusetts Institute of Technology, Cambridge, Massachusetts 021-42, 1993. The MIT Press.
- [8] M. Codish and B. Demoen. Deriving polymorphic type dependencies for logic programs using multiple incarnations of prop. *Lecture Notes in Computer Science*, 864:281–297, 1994.
- [9] M. Codish and B. Demoen. Analysing logic programs using “prop”-ositional logic programs and a magic wand. *Journal of Logic Programming*, 25(3):249–274, December 1995.
- [10] A. Cortesi and G. Filé. Abstract interpretation of logic programs: an abstract domain for groundness, sharing, freeness and compoundness analysis. In *Proceedings of the Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 52–61, New Haven, Connecticut, USA, 1991.
- [11] A. Cortesi, G. Filé, and W. Winsborough. Optimal groundness analysis using propositional logic. *Journal of Logic Programming*, 27(2):137–168, 1996.
- [12] P. Cousot and R. Cousot. Abstract interpretation: a unified framework for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the fourth annual ACM symposium on Principles of programming languages*, pages 238–252. The ACM Press, 1977.
- [13] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the sixth annual ACM symposium on Principles of programming languages*, pages 269–282. The ACM Press, 1979.
- [14] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(1, 2, 3 and 4):103–179, 1992.
- [15] Philip W. Dart. On derived dependencies and connected databases. *Journal of Logic Programming*, 11(2):163–188, August 1991.
- [16] K. Furukawa, editor. *Proceedings of the Eighth International Conference on Logic Programming*. The MIT Press, 1991.
- [17] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Evaluation of the Domain PROP. *Journal of Logic Programming*, 23(3):237–278, June 1995.
- [18] K. Horiuchi and T. Kanamori. Polymorphic type inference in Prolog by abstract interpretation. In K. Furukawa, H. Tanaka, and T. Fujisaki, editors, *Proceedings of the Sixth Conference on Logic Programming*, pages 195–214, Tokyo, June 1987.
- [19] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 154–165, Cleveland, Ohio, USA, 1989.

- [20] D. Jacobs and A. Langen. Static analysis of logic programs for independent and parallelism. *Journal of Logic Programming*, 13(1–4):291–314, 1992.
- [21] N.D. Jones and H. Søndergaard. A semantics-based framework for abstract interpretation of Prolog. In S. Abramsky and C. Hankin, editors, *Abstract interpretation of declarative languages*, pages 123–142. Ellis Horwood Limited, 1987.
- [22] T. Kanamori. Abstract interpretation based on Alexander templates. *Journal of Logic Programming*, 15(1 & 2):31–54, January 1993.
- [23] T. Kanamori and T. Kawamura. Abstract interpretation based on oldt resolution. *Journal of Logic Programming*, 15(1 & 2):1–30, January 1993.
- [24] A. King. A synergistic analysis for sharing and groundness which traces linearity. *Lecture Notes in Computer Science*, 788:363–378, 1994.
- [25] B. Le Charlier and P. Van Hentenryck. Groundness analysis of Prolog: implementation and evaluation of the domain Prop. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 99–110. The ACM Press, 1993.
- [26] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
- [27] L. Lu. Abstract interpretation, bug detection and bug diagnosis in normal logic programs. PhD thesis, University of Birmingham, 1994.
- [28] L. Lu. Type analysis of logic programs in the presence of type definitions. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based program manipulation*, pages 241–252. The ACM Press, 1995.
- [29] L. Lu. A polymorphic type analysis in logic programs by abstract interpretation. *Journal of Logic Programming*, 36(1):1–54, 1998.
- [30] K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems*, 2(1–4):181–196, 1993.
- [31] K. Marriott, H. Søndergaard, and N.D. Jones. Denotational abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems*, 16(3):607–648, May 1994.
- [32] C. Mellish. Some global optimisations for a Prolog compiler. *Journal of Logic Programming*, 2(1):43–66, 1985.
- [33] C. Mellish. Abstract interpretation of Prolog programs. In S. Abramsky and C. Hankin, editors, *Abstract interpretation of declarative languages*, pages 181–198. Ellis Horwood Limited, 1987.
- [34] K. Muthukumar and M. Hermenegildo. Combined determination of sharing and freeness of program variables through abstract interpretation. In Furukawa [16], pages 49–63.
- [35] K. Muthukumar and M. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *Journal of Logic Programming*, 13(1, 2, 3 and 4):315–347, 1992.
- [36] U. Nilsson. Towards a framework for the abstract interpretation of logic programs. In P. Deransart, B. Lorho, and J. Maluszynski, editors, *Proceedings of the International Workshop on Programming Language Implementation and Logic Programming*, pages 68–82. Springer-Verlag, 1988.
- [37] R. A. O’Keefe. Finite fixed-point problems. In J.-L. Lassez, editor, *Proceedings of the fourth International Conference on Logic programming*, volume 2, pages 729–743. The MIT Press, 1987.
- [38] H. Søndergaard. An application of abstract interpretation of logic programs: occur check problem. *Lecture Notes in Computer Science*, 213:324–338, 1986.

- [39] H. Søndergaard. Immediate fixpoints and their use in groundness analysis. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science 1180, pages 359–370. Springer, 1996.
- [40] L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, 1986.
- [41] R. Sundararajan and J.S. Conery. An abstract interpretation scheme for groundness, freeness, and sharing analysis of logic programs. In R. Shyamasundar, editor, *Proceedings of 12th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 203–216. Springer-Verlag, 1992.